Please amend page 17, lines 2 - 3 of the specification, with the amended paragraph as set forth in this paper at page 8.

Please amend page 17, lines 5 - 7 of the specification, with the amended paragraph as set forth in this paper at page 9.

Please amend page 17, lines 17 - 18 of the specification, with the amended paragraph as set forth in this paper at page 10.

Please amend page 17, line 24 of the specification, with the amended paragraph as set forth in this paper at page 11.

Please amend page 18, line 20 through page 19, line 5 of the specification, with the amended paragraph as set forth in this paper at page 12.

Please amend page 40, line 25 through page 41, line 8 of the specification, with the amended paragraph as set forth in this paper at page 13.

Please amend page 59, lines 16 - 19 of the specification, with the amended paragraph as set forth in this paper at page 14.

Please amend page 61, lines 5 - 11 of the specification, with the amended paragraph as set forth in this paper at page 15.

Please amend page 64, lines 11 - 15 of the specification, with the amended paragraph as set forth in this paper at page 16.

Please amend page 64, line 16 through page 65, line 1 of the specification, with the amended paragraph as set forth in this paper at page 17.

Please amend page 83, line 29 through page 84, line 4 of the specification, with the amended paragraph as set forth in this paper at page 18.

Please amend page 90, lines 13 - 17 of the specification, with the amended paragraph as set forth in this paper at page 19.

Please amend page 123, lines 9 - 18 of the specification, with the amended paragraph as set forth in this paper at page 20.

Please amend page 125, lines 16 - 19 of the specification, with the amended paragraph as set forth in this paper at page 21.

Please amend page 125, line 20 through page 126, line 2 of the specification, with the amended paragraph as set forth in this paper at page 22.

Please amend page 128, line 14 through page 129, line 4 of the specification, with the amended paragraph as set forth in this paper at page 23.

Please amend page 129, lines 5 - 9 of the specification, with the amended paragraph as set forth in this paper at page 24.

Please amend page 129, lines 10 - 19 of the specification, with the amended paragraph as set forth in this paper at page 25.

Please cancel the abstract of the specification as filed and amend the abstract of the specification with the new abstract as set forth in this paper at page 26.

In the claims:

Cancel claims 1 – 57. Please add the new claims as set forth in this paper beginning at page 27.

RESPONSE

Response:

Applicant's response and associated remarks begin on page 37 of this paper.

NOTICE REGARDING COPYRIGHTED MATERIAL

A portion of the disclosure of this patent document contains material which is subject to copyright protection. The copyright owner has no objection to the facsimile reproduction by anyone of the patent document or the patent disclosure as it appears in the file or records as maintained by the United States Patent and Trademark Office, but otherwise reserves all copyright rights whatsoever.

There are many styles of programming language grammars in use today. Two of these styles are brought together into the programming language grammar of the present invention. The first style encompasses all languages which derive from the C programming language, introduced some 20 years ago, including ~~Java~~ Java™, C++, and JavaScript, and now, the grammar described herein. The second style includes a large group of languages which derive their usefulness by offering the programmer the ability to create "regular expressions" for searching, recognizing, and tokenizing text. The grammar of the present invention combines these two styles by making regular expressions a primitive data type of a C-style language.

The table of Figure 3 of section 7 breaks down each of the above components into a list of actual source code ~~Java~~ Java™ files, which are provided on CD. A few source files of the invention are not listed in the table under any of the five components because they are simple utilities, such as growable arrays of chars, growable arrays of objects, and so forth.

Detailed source code files on CD-ROM, Copies 1 and 2, are provided to this specification. The source code should be consulted to provide details of a specific embodiment of the invention in conjunction with the discussion of the routines of the methods and systems in this specification. The source code in CD-ROM, copies 1 and 2, implements the present invention including architecture and dataflow.

String literals, character literals, and integer literals are expressed as in ~~Java~~ Java™; accordingly, the *char* datatype is a 2-byte Unicode character.

Arrays are defined and used as in ~~Java~~ Java™, except that arrays are automatically growable; there is no such thing as an ArrayIndexOutOfBoundsException in the grammar of this invention.

Casting from one type to another uses the syntax that looks like a function call, not the syntax adopted by ~~Java~~ Java™.

Static blocks of code look and behave exactly as in ~~Java~~ Java™.

The language of the present invention has created a primitive data type called *Pattern* that is used to represent every instantiation of a regular-expression. Thus, regular expressions can be assigned to scoped variables of type *Pattern*, allowing regular expressions to be built both dynamically and incrementally, as seen in examples. For a programmer using the grammar, *Pattern* is to be thought of as a primitive, not a structured object. Or, equivalently, the programmer may consider it as a complex structure, so long as he/she realizes that it is really a *const* structure, similar in this respect to the semantics of a ~~Java~~ Java™ *String*, as known in the art. Once a regular expression has been created and assigned to a variable of type *Pattern*, this variable (plus any other variables to which it is assigned) will always reference that "particular" regular expression object, semantically and in fact.

Many regular expression grammars do not have a concatenation operator at all. In such grammars of the art, concatenation is assumed as sub-expressions read from left-to-right. However, in the grammar of this invention, which adopts the C-expression syntax for regular expressions, white-space is not acceptable as a way to separate terms of a concatenation, because white-space is not an operator of C. The primary concatenation operator is the + operator, chosen because this is also the operator used to express String concatenation, as with ~~Java~~ Java™. There are two additional concatenation operators in the grammar, >+ and <+ signifying left-to-right concatenation and right-to-left concatenation respectively. The + operator, when used for concatenation of Patterns is referred to as the "default" concatenation operator, and always carries the "left-to-right" connotation. The syntax for the 3 binary concatenation expressions is as follows:

all instructional side-effects (instruction-arcs of corresponding NFA) of the secondary are ignored, that is, effectively removed; i.e. only the "eating"/"matching" characteristics of the ~~primary~~ secondary are considered in the convolution; and

The String grammar composition is quite simple. A String literal of this grammar (whose syntax matches that of a ~~Java~~ Java™ String literal), or alternatively a declared variable of type String, when used in a composition, is automatically widened to a Pattern, and can be used in any Pattern composition. As such, the widened String-Pattern is treated exactly as a concatenation of its individual characters (as seen in StringGrammar.java). Here the "right-to-left" or "left-to-right" sense does not matter because there are no instruction-arcs implied within a String. Therefore, the following line of code:

This section starts with simple examples which show that the grammar covers all of the regular expression forms expected by a regular expression programmer. The section then moves to increasingly more complicated expressions, showing the novelty of the grammar of the present invention. Because the grammar of the present invention adheres so closely to the form of C and to a certain extent ~~Java~~ Java™, such a programmer will easily read and understand the example scripts.

The following examples do not include all of the standard grammar forms offered by the C, C++, and ~~Java~~ Java™ programming languages. This is not an implied limitation of the grammar of this invention, nor its engine, for standard grammar forms not shown (or implemented) – such as an *include* statement, an *extern* keyword, global/static variables, packages, class and member function definition – can be implemented in the interpreter and VM of this invention through techniques known in the art. The examples will show that the grammar of this invention has: (1) introduced novel forms of regular expressions which are not easily or not at all reducible to grammar forms already seen in the art; (2) adopted the C-style of expressions (such as for arithmetic expressions), including C's precedence and associativity rules, and has applied this expression syntax to the regular expressions of this invention.

The "pre-list" and "post-list" are comma-separated statement lists, as required also by the standard "for-statement" grammar. However, because these lists serve as the only way to embed statements into the "do-pattern", most restrictions regarding the usage of statements within these lists have been relaxed, relative to the standard restrictions enforced by C, ~~Java~~ Java™, and C++ compilers. The grammar of the present invention specifies that any statement can be used within these statement lists except the *tokenize* statement and a function *return* statement. Multiple variables of different types can even be declared, but these declarations cannot combine multiple variables into the same statement (because comma is already being used as a statement separator).

Next, let the engine execute the NFA corresponding to the ~~subjunctive~~ secondary expression against the same input "string", but ignoring all instruction-arcs and arc-numbers – here the engine simply saves all unique accept states where an accept state is defined solely by number of characters "eaten" – this set is labeled SecondarySolutionSet.

The implementation of this engine uses a virtual machine architecture. The primary motivation for this requirement will be explained in section 8.5.1.1 below. The Virtual Machine (VM) selected for the current implementation of this invention uses "FORTH-like" instructions, sharing similarities also with the ~~Java~~ Java™ Virtual Machine. Although the term "FORTH" is used to describe the VM of this invention, it does not imply a "standard" FORTH engine and grammar. Rather, the term FORTH here refers to an approach for building virtual machines and their grammars. In this sense many languages in use can be likewise called "FORTH-based", even though their atomic dictionary of FORTH words may differ extensively. PostScript is an example of such a language. FORTH-like languages share some important features, which can be inferred from the following list of FORTH-like aspects of the Virtual Machine of this invention:

"global" variables. The way that the C-like global variables (called static in ~~Java~~ Java™) of the language of this invention are modeled is by creating a VM construct in the FORTH engine that is an array for globals, and using a global literal and creating a syntax for global variable literals – $$0, $$1, $$2, etc.

"local" variables. The way that the C-like local variables ~~(called static in Java)~~ of the language of this invention are modeled is by creating a VM construct in the FORTH engine for a stack frame, and using a "local" literal and creating a syntax for local variable literals – $0, $1, $2, etc. This is much like the ~~Java~~ Java™ VM, and is in contrast to the standard FORTH variants which have the concept of a return stack". "Local variable" literals are also used for variables defined within the scope of a high-level "do-pattern".

Although a VM "assembly" language which includes primitive instructions for manipulating the Pattern type is seen as having advantages, it is not a requirement based on the above arguments. The next problem to consider is that of "do-pattern" expressions. If the grammar of the present invention were to be incorporated into either C++ or ~~Java~~ Java™, the "do-pattern" pre-list and post-list statements would be compiled to native machine instructions (C++) or VM instructions (~~Java~~ Java™), similar to what is done in the present invention. These instructions would reference and manipulate frame variables and register variables (native and VM respectively). These instructions could also reference built-in (primitive) container objects, such as Strings and Patterns and Character-Class objects, a requirement that makes a strong argument for a VM architecture such as ~~Java~~ Java™ 's versus C++. But a very big obstacle for a C++ architecture is that these "do-pattern" instruction arrays are (and must be) treated as data arrays from the point at which they are created, until the snippets "accepted" by the automata's execution are then assembled into one long array of instructions. C++ architectures generally allow only the interpreter to create code segments. It is seen that the internal functions executing the automata would not normally have "privileges" to dynamically create and call a "code-segment" based on the "do-pattern" ACCEPT fragments.

This latter obstacle favors a VM architecture such as that of ~~Java~~ Java™, which could be more easily adapted to the unique requirements of the regular expression grammar of this invention. Nevertheless, it is expected that both ~~Java~~ Java™ 's VM instruction set, and its current engines on the market would have to be extended to solve the aforementioned problem of allowing arrays of VM instructions to be assembled as data and then executed.

Yet another hurdle for incorporating the regular expression grammar of this invention into either the ~~Java~~ Java™ Virtual Machine or into native C++ compilers would be that of modeling instantiated production rules. It will be seen in section 8.5.1.3 below that in the VM of the present invention, instantiated production rules resolve to low-level "do-pattern" expressions. That is not a problem. However, the present embodiment also requires an additional frame construct in the VM, the "template frame", to allow the binding of production rule parameters to the instantiation object (a requirement also explained in section 8.5.1.3), and to allow nested rule instantiations. The concept of a "template-frame" exists in neither the present ~~Java~~ Java™ VM, nor current native machine architectures. This aspect of production rule modeling favors a virtual machine architecture, which by the "virtual" aspect of a VM, allows the creation of any new VM construct needed.

Method and system of integrating side-effects into regular expressions. A grammar form allows capturing the match to a regular expression into a named variable, or String reference, so long as the formulation is in scope. Another grammar form provides the ability to wrap side-effect producing statements around regular expressions. Another grammar form provides the ability to create named, reusable production rules as encapsulations of regular expressions, such encapsulations also being parameterizable. The parameters can be used in matching characteristics and/or side-effect characteristics of the rule. Another grammar form is the subjunctive, and its ability not only to narrow the specificity of the primary term, but also to uphold the side-effects implied by the primary term. In all cases, side-effects are triggered as instructions in the virtual machine if and only if ultimately associated with the traversal of the "winning thread" through the automata.

58. (new) A computer-implemented method of integrating side-effects into regular expressions, the method comprising the steps of:

defining a built-in datatype, to represent any regular expressions, or regex composition, in such a way that a regular expression referenced by a variable of such datatype is immutable, and wherein the datatype facilitates composition using variables, and allows enforcement of strong type compatibility for regular expressions, such as when and where regular expressions can be used in the overall grammar of a programming language;

defining a grammar for standard regular expression compositions, such as character-classes, string and character literals, a wildcard, repeats, iterates, concatenations, and unions;

defining a grammar for general regular expression compositions that contribute side-effects;

defining a grammar for expressing the particular side-effect of "capture";

defining a grammar for regex compositions in which side-effects are inhibited;

defining a grammar for creating named, re-usable encapsulations for regular expressions, wherein the encapsulations also allow parameterization and the integration of side-effects; and

defining a grammar for a binary composition, called the "subjunctive", including a primary and secondary term, in which the secondary term is used to narrow the specificity of the primary (by qualifying its matches), while allowing the side-

effects of the primary to trigger in the expected way;

wherein side-effects are encoded as general statements within the same language or grammar in which the regular expressions are composed;

wherein the side-effects are executed by the same machine or virtual machine that executes all of the translated instructions of the language hosting the regular expression;

wherein the side-effects are executed as a step of the matching of a regular expression to data, the step occurring just after the "winning" match has been determined, and just before the next statement in code outside of the regular expression is executed;

wherein the side-effects do not alter the matching characteristics of a regular expression composition or sub-composition;

wherein the side-effects are triggered if and only if the regex composition or sub-composition associated with its side-effects is ultimately matched to data; and

wherein the method allows solutions to data matching problems to encode a greater portion of the solution within regular expressions and a lesser portion of the solution within surrounding code.

59. (new) The method of claim 58, wherein a regex composition grammar form (termed the "capture-pattern") allows the match to a regex sub-composition not only to be captured, but to be captured into any named variable (of type *String*) available in the scope in which its

matchable *Pattern* sub-composition is defined.

60. (new) The method of claim 59, wherein the "capture-pattern" syntax further allows the capture reference to be any valid reference of type *String*, in which the reference expression's elements are available in the scope in which the capture-pattern's matchable *Pattern* sub-composition is defined.

61. (new) The method of claim 58, wherein a regex composition grammar form (termed the "do-pattern") allows side-effect producing statements to be wrapped around a matchable *Pattern* sub-composition, wherein the "do-pattern" syntax consists of a suitable keyword signaling its start, such as *do*, followed by a pre-list of statements, followed by a *Pattern* sub-composition which determines what the "do-pattern" matches, followed by a post-list of statements, the pre-list and post-list statements comprising statements within the same programming language in which regular expressions are composed.

62. (new) The method of claim 61, wherein the pre-list and post-list statements of the "do-pattern" comprise any of the statement grammar forms of the programming language of the invention, except for the *return* statement and the *tokenize* statement.

63. (new) The method of claim 61, wherein the "do-pattern" establishes its own scope, such that temporary variable(s) are normally defined in the pre-list, and are usable (once defined) in the pre-list, and/or the matchable *Pattern* sub-composition, and/or the post-list.

64. (new) The method of either claims 59 or 61, wherein "do-patterns" and "capture-patterns" involved in a match and/or sub-match associate side-effect instructions to the matching process, wherein the instructions are automatically executed if and only if the "do-

patterns" and "capture-patterns" are involved in the "winning" match; wherein the instructions are a direct translation of pre-list and post-list statements in a "do-pattern"; and wherein the instructions are generated for a "capture-pattern" by modeling the "capture-pattern" as a low-level "do-pattern".

65. (new) The method of either claims 59 or 61, wherein "do-pattern" and/or "capture pattern" side-effects selected for execution along with the "winning" match are determined unambiguously based on a set of ambiguity resolution rules appropriate (in the presence of side-effects) to unions, concatenations, repeats, and iterates.

66. (new) The method of claim 58, wherein a binary regex composition, called the "subjunctive", has two forms, *but* and *butnot*, both of which consist of a primary term and a secondary term, wherein the primary term contributes not only a primary matching characteristic, but also side-effects, wherein the secondary term contributes no side-effects (because they are automatically inhibited), and wherein the secondary term is used to restrict the possible matches of the primary term.

67. (new) The method of claim 66, wherein a suitably named subjunctive keyword, such as *but*, signals that the secondary term restricts the matches of the primary to the extent that the secondary is able to match the same data matched by the primary at the same time, and wherein the secondary term in no way changes the side-effects implied by the primary's match (assuming the primary is allowed by the secondary to match).

68. (new) The method of claim 66, wherein a suitably named subjunctive keyword, such as *butnot*, signals that the secondary term restricts the matches of the primary to the extent that

the secondary is **not** able (in any way) to match the same data matched by the primary at the same time, and wherein the secondary term in no way changes the side-effects implied by the primary's match (assuming the primary is allowed by the secondary to match).

69. (new) The method of claim 58, wherein regular expressions can be encapsulated as named, re-usable production rules, the rules having the characteristic of parameterizability, the rules comprising bodies which can incorporate side-effect producing regex compositions, such as "do-patterns" and "capture-patterns".

70. (new) The method of claim 69, wherein the production rule grammar permits the parameterizability of side-effects and/or matching characteristics.

71. (new) The method of claim 70, wherein "in", "out" and "in out" parameters of a production rule can be used to parameterize the side-effects of the rule, wherein such parameters are usable to formulate the capture-reference of a "capture-pattern" (if such a "capture-pattern" is found within the rule's body) , and wherein such parameters are usable in the formulation of the pre-list and/or post-list statements of a "do-pattern" (if such a "do-pattern" is found within the rule's body).

72. (new) The method of claim 70, wherein parameters of the "in" variety can be used to parameterize the matching characteristics of the rule.

73. (new) The method of claim 72, wherein a parameter of the "in" variety can be declared of type *Pattern*, and used not only to allow its rule to build upon the matching characteristics of the actual *Pattern* object being passed/bound, but also to allow its rule to inherit the side-effect characteristics of the actual *Pattern* parameter being passed/bound; wherein the

parameter is deemed a target of the rule, consistent with the "find-first" and "find-nth" design patterns.

74. (new) The method of claim 69, wherein the association of actual parameter values (or references to variables) to a signature compatible production rule results in an "instantiation" object that is type-compatible with the *Pattern* datatype, and can therefore be used anywhere a *Pattern* object can be used.

75. (new) The method of claim 58, whereby a single (suitably composed) production rule encapsulates the entire problem of directed capture of multi-dimensional data, leveraging "do-patterns" and "capture-patterns", and leverages the ability of production rule parameters to be used in the "do-patterns" and "capture-patterns".

76. (new) The method of claim 58, wherein production rules that have a base-case which must not match any data are defined based on the use of a conditional operator and the *reject* literal, wherein the *reject* literal serves as a base case.

77. (new) The method of claim 58, wherein an *inhibit* keyword transforms any *Pattern* object to a new *Pattern* object that inherits all of the original's matching characteristics and none of the original's side-effect characteristics.

78. (new) A computer system for interpreting or compiling and then executing a programming language with support for regular expressions, including a plurality of support for integrating side-effects into regular expressions, the system comprising:

a computer processor ;

the programming language and regex grammar of claim 58;

a computer program which executes computer program modules/scripts written in the programming language and, while the computer program is being hosted by the computer processor, the computer program is also the host for the computer program modules/scripts which are its target executables;

a data input device providing a user-defined program module/script (written by user according to the grammatical rules of the programming language) to the host computer program;

a data input device providing a source of data to be processed by the regular expressions of a program module/script;

a set of virtual instructions, into which instructions the program modules/scripts are translated, and which instructions are executed by a subcomponent of the host computer program called the "virtual machine"; and

a subset of virtual instructions, which map accessibility functions of input/output devices in the system to instructions, so that program modules/scripts can access input/output devices and thereby provide the programmer with output consistent with the user defined program module/script.

79. (new) The system of claim 78, further comprising support (in the "host computer program") for regular expressions and support for integrating side-effects into regular expressions by:

extending the virtual machine, including its instruction set, into which not only the normal statements and function calls of the programming language can be defined, but also its regular expression compositions, including also instructions that execute regular expressions against data;

defining a set of regular expression composition instructions that allow every regular expression of the programming language to be modeled in the virtual machine as an immutable object;

implementing a group of subset construction techniques that permit the transformation of every regular expression object in the virtual machine into a form suitable for execution against data;

implementing a runtime automata execution technique that allows the execution of all of the regular expressions of the invention, once transformed into automata, against data, in such a way that each character from the stream is examined at most once;

implementing the runtime automata execution technique in such a way that side-effects of the automata are accumulated as instructions to a "winning" thread, and executed if and only if the regular expressions to which they correspond are involved in a "winning" match against the data;

implementing the compositions and subset constructions of regular expressions and their automata with inclusion of both instruction-arcs to model side-effects as well as arcnum-sequences to serve as a criteria for

unambiguously selecting a "winning" match to data, in the presence of side-effects; and

implementing the runtime automata execution technique in such a way that a set of "automata-threads" is maintained, that represent the set of all ways in which data not only can be matched, but also all ways in which side-effects can be accumulated by the ultimate "winning" thread.

80. (new) The system of claim 79, wherein side-effects are modeled as instruction-arcs, which are placed into composition graphs (automata) to fully model both "do-patterns" and "capture-patterns," wherein the instruction-arcs imply the existence of arcnum-sequences that must be attached to "eating" arcs of the automata in order to allow the runtime automata execution technique to make decisions on automata-thread preference.

81. (new) The system of claim 79, further comprising a subset construction technique that allows a modeling automata/graph to be converted into a form suitable for execution, including the presence of both instruction-arcs and arcnum-sequences.

82. (new) The system of claim 79, further comprising a subset construction technique that allows a subjunctive convolution to be performed to represent a subjunctive expression, resulting in yet another composition graph that models semantics of the subjunctive expression by accurately representing both possible matches and possible side-effects of the subjunctive expression.

83. (new) The system of claim 79, wherein the runtime automata execution technique includes a pruning step to limit proliferation of the automata threads to be tracked, thus

preventing an exponential explosion of automata threads in the presence of nested, repeating side-effect compositions.